

# Automatic Parameter Recommendation for Practical API Usage

Cheng Zhang<sup>1</sup>, Juyuan Yang<sup>2</sup>, Yi Zhang<sup>2</sup>, Jing Fan<sup>2</sup>, Xin Zhang<sup>2</sup>, Jianjun Zhao<sup>1, 2</sup>, Peizhao Ou<sup>2</sup>

<sup>1</sup>Department of Computer Science and Engineering, <sup>2</sup>School of Software

Shanghai Jiao Tong University, China

{cheng.zhang.stap, xiaoyangmie, zorozy, louievan, hyperzh, zhao-jj, mads}@sjtu.edu.cn

**Abstract**—Programmers rarely build software from scratch; instead they extensively leverage existing libraries and frameworks by using the exposed application programming interfaces (APIs). However, correctly and efficiently choosing and using APIs from unfamiliar libraries is still a non-trivial task. Programmers often need to ruminate API documentations (that is often incomplete) or inspect existing code examples (that is often in absent) to understand its usage patterns. Recently, various techniques have been proposed to alleviate this problem by creating API summarizations, mining code examples, or showing common API call sequences. However, few technique focuses on recommending correct API parameters.

In this paper, we propose an automated technique, called Precise, to address this problem. Differing from common recommendation systems, Precise mines existing code base, uses an abstract usage pattern representation for each API usage scenario, and then builds a parameter usage database. Upon on request, Precise queries the database for abstract usage patterns in similar contexts and generates parameter candidates by concretizing the patterns adaptively.

We evaluated our technique on several real-world large-scale programs. The results show that our technique is more general and applicable than existing recommendation systems, specially, 64% of the API parameter recommendations are useful and 53% of the recommendations are exactly the same as the actual parameters needed. We also performed a user study to show our technique is useful in practice.

**Keywords**—recommendation; API; parameter; code completion;

## I. INTRODUCTION

Application programming interfaces (APIs) are extensively used in modern software development in order to reuse libraries and frameworks. Since there exist numerous APIs providing various functionalities, developers are likely to encounter unfamiliar APIs in their daily work. Unfortunately, APIs are difficult to learn in general, due to various factors, such as intrinsic complexity of the application domain or inadequate API design [20]. As a result, developers usually have to make considerable effort to learn how to use an API correctly and efficiently.

To alleviate this problem, a number of techniques [10], [21], [15] have been proposed to facilitate the usage of APIs. Among these, a highly automated approach is a code completion system, which promptly provides developers with programming suggestions, such as which methods to call and which expressions to use as parameters. Traditional code completion systems generally propose their suggestions

based on type compatibility and visibility. Such suggestions may become insufficient for complex frameworks, where some types have too many member methods or fields to suggest. Therefore, some recent work [8] tries to improve the suggestions via mining API usage data from code bases.

Existing techniques are mostly focused on *choosing the right method to call*. Nevertheless, from our own programming experience and as discussed in [8], it is a non-trivial task to *choose the right (actual) parameter(s) for a method call* in an API usage. Table I shows the statistics of the API declarations and usage in the code bases of Eclipse 3.6.2 [3], Tomcat 7.0 [1], and JBoss 5.0 [5].

Table I  
STATISTICS ON METHOD DECLARATIONS AND INVOCATIONS

Program	param declaration	non-param declaration	param invocation	non-param invocation
Eclipse 3.6.2	64%	36%	57%	43%
JBoss 5.0	58%	42%	60%	40%
Tomcat 7.0	49%	51%	60%	40%
average	<b>57%</b>	43%	<b>59%</b>	41%

<sup>1</sup>“param” and “non-param” are abbreviations for “parameterized” and “non-parameterized”, respectively.

From Table I, we can see that 57% of the method declarations are parameterized, that is, the methods need to be passed one or more parameters when they are called. In accordance with the statistics of method declarations, 59% of the method invocations actually have parameters. Furthermore, over 50% of these actual parameters cannot be recommended by existing code completion systems, because the expected parameters are too complex to recommend<sup>2</sup>. In some cases, even if the existing code completion systems can provide correct recommendations, it can be quite difficult to find the right parameter from a large number of parameter candidates. Figure 1 shows an example of calling an API in the Eclipse org.eclipse.help.ui project.

In this example, the developer has to choose the right actual parameter (i.e., `IHelpUIConstants.TAG_DESC`) from over **50** string variables and constants. Unfortunately, the field `TAG_DESC` is even not visible in Figure 1, as it is ranked too low by dictionary order. More importantly, since the code completion system does not suggest that the class `IHelpUIConstants` should be used, the developer

<sup>2</sup>We use the Eclipse JDT code completion system as a typical example. The details will be explained in Section III-A

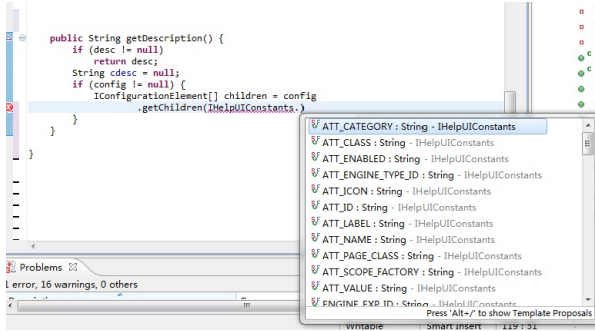


Figure 1. An example of parameter recommendation by the default Eclipse JDT completion system.

probably has to learn this usage from code examples or documentation. Therefore, it would be really helpful if the code completion system could provide more informative recommendations of API parameters and rank them in a more sophisticated way (e.g., suggest using `IHelpUIConstants` and put `TAG_DESC` near the top of the list).

Besides slowing down the development process, unfamiliarity with parameter usage may even harm the correctness of programs. Bug 153895 in the bug database of Eclipse<sup>3</sup> is a typical example of the bugs caused by incorrect usage of API parameters. The bug reporter suggested that invocations to method `getBestActiveBindingFormattedFor` of class `IBindingService` should be used as the actual parameters of two method invocations. In the end, the developer took the suggestion and fixed the bug by replacing the incorrect parameters with the correct ones. It is worth noting that existing code completion systems can provide little help in this case, because such method invocations are too complex to recommend (also see Section III-A).

In this paper, we propose *Precise*, an automated approach to parameter recommendation for API usage, which is able to recommend the kinds of API parameters that are frequently used in practice but are mostly overlooked by existing code recommendation systems. During programming, when the developer has already selected a method and is about to determine the actual parameter(s) of that method, *Precise* automatically recommends a concise list of parameter candidates that are well sorted. The basic idea of *Precise* is to learn usage patterns from existing programs and adaptively recommend parameters based on these patterns and the current context. On receiving a request for a recommendation, *Precise* first uses *k*-Nearest Neighbor (kNN) [23] queries on the usage database built from a large code base to find abstract usage patterns of parameters that are commonly used in similar contexts. Then *Precise* generates concrete recommendations based on the abstract usage patterns. *Precise* ranks its recommendations with respect to the similarity and frequency of usage, trying

to help the programmer select the right parameters more easily. Two heuristic rules are used to reduce the search space of parameters, making *Precise* practical and efficient. The results of an objective experiment show that *Precise* is able to recommend parameters with good accuracy under strict constraints on the total number of recommendations. Specifically, the exact expected parameters are included in the top 10 recommended parameters in 53% of the cases, and 64% of the recommendations can provide useful information to help choose the right parameters.

The main contributions of this paper are:

- 1) *Precise*, an automatic parameter recommendation technique, to improve existing code completion systems. To the best of our knowledge, it is the first automatic technique focused on parameter recommendation.
- 2) We have implemented *Precise* as an Eclipse plug-in and performed an evaluation to show the effectiveness of *Precise*. Combined with the Eclipse JDT, our plug-in can provide useful parameter recommendations in over 72% of cases. We have also conducted a user study to confirm the usefulness of *Precise*.

The rest of this paper is organized as follows. Section II shows the result of an empirical study on the usage of API parameters. Section III describes the technical details of *Precise*. Section IV shows the results of the objective experiment and user study on *Precise*. Section V compares *Precise* with closely related work. Section VI concludes this paper and discusses our future work.

## II. API PARAMETER USAGE IN PRACTICE

A major challenge in parameter recommendation is that there are generally too many possible parameter candidates that are type compatible with the expected parameter. For example, when we try to recommend the actual parameter for the method `m(int a)`, there are an almost infinite number of parameter candidates (of type `int`), including all the integer literals<sup>4</sup>, accessible integer variables, method calls returning integers, etc. In addition, the parameter can be an arithmetic expression, such as `(a + b * c)`, which may have a very complex structure. Therefore, as a fundamental strategy of *Precise*, we propose two heuristic rules to focus the approach on parameters of a limited number of structure patterns. The underlying assumption is that *most of the parameters fall into a small number of categories of structural patterns*.

In order to support the assumption, we have done an empirical study to see how API parameters are used in practice by looking into three subjects, namely Eclipse 3.6.2, JBoss 5.0, and Tomcat 7.0. These are large-scale programs that are widely used in real-world settings. Thus the result of our study is supposed to be reasonably representative.

<sup>3</sup>[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=153895](https://bugs.eclipse.org/bugs/show_bug.cgi?id=153895)

<sup>4</sup>For example, in Java, the integer literals are the integers, such as 1, 2, and 100, in the interval `[Integer.MIN, Integer.MAX]`.

As shown in Table II, over 89% of the actual parameters of APIs can be classified into 7 categories of expression types. The expression types used in Table II are formally defined in Eclipse’s JDT documentation [2] that is in line with the Java language specification [4]. We just give a brief description of the categories of expression types below:

- 1) A *simple name* is a Java identifier which is not a keyword, boolean literal or null literal. It can be an unqualified class name, abbreviated field access, etc.
- 2) A *method invocation* is an expression to invoke a method with or without a base expression. Its general form is `base.methodName(arguments)`, where `base` and `arguments` are optional.
- 3) A *field access* is an expression to access a field of a class or an instance. Its general form is `base.fieldName`, where `base` is necessary. Note that an access to a field of the current class (or super types) may have no `base` part. In this case, it is considered as a *simple name* rather than a field access.
- 4) A *qualified name* resembles a field access and its general form is `qualifier.name`. Common qualified names include fully qualified classes and some kinds of field accesses (e.g., `foo.bar`, but not `foo().bar`).
- 5) An *array access* is an expression to access an element of an array. Its general form is `array[index]`.
- 6) A *cast expression* is an expression to cast one type to another. Its general form is `(type)expression`.
- 7) *Literals* include string literals, number literals, character literals, type literals, boolean literals, and null literals.

Table II  
STATISTICS ON EXPRESSION TYPES OF ACTUAL PARAMETERS.

Expression Type	Eclipse 3.6.2	JBoss 5.0	Tomcat 7.0	average
simple name	47.35%	38.79%	39.97%	42.04%
boolean literal	3.83%	2.05%	2.13%	2.67%
null literal	1.81%	1.67%	1.72%	1.73%
method invocation	12.16%	11.96%	10.44%	11.52%
qualified name	11.28%	4.33%	4.28%	6.63%
field access	1.15%	0.57%	0.32%	0.68%
array access	1.58%	1.09%	0.63%	1.10%
string literal	4.71%	21.71%	19.74%	15.39%
number literal	3.44%	5.66%	4.04%	4.38%
character literal	1.02%	1.29%	0.56%	0.96%
type literal	0.42%	0.68%	2.24%	1.11%
cast expression	1.16%	1.28%	0.37%	0.94%
total percentage	89.91%	91.08%	86.44%	<b>89.14%</b>

There exist several expression types out of the scope of the 7 categories, including array creation, assignment, infix expression, etc. They together make up 11% of the actual parameters used in the subjects.

### III. APPROACH

Figure 2 shows the workflow of Precise which consists two phases. In phase 1, Precise builds a parameter usage database by analyzing parameter usage instances and their

contexts in a code base. In phase 2, for a specific recommendation request for a program under development, Precise queries the database using the context of the request as the key and retrieves a group of abstract usage patterns. Then Precise concretizes and sorts the recommendations. Both phases of Precise are governed by two heuristic rules that make the technique practical while still being effective. In this section, we first introduce the rules and then describe the two phases in detail.

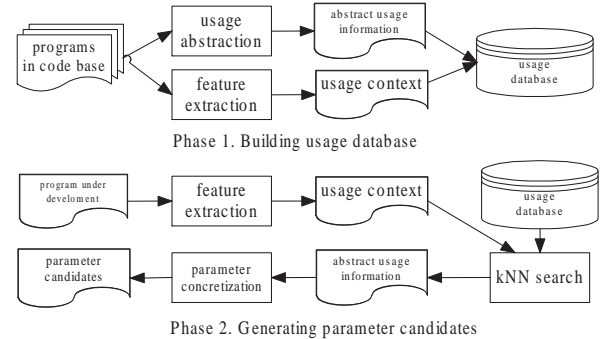


Figure 2. The overall workflow of Precise.

#### A. Heuristic Rules for Search Space Reduction

**Rule of expression type.** The first rule is to restrict the two phases to the parameters of certain expression types, namely method invocation, field access, qualified name, array access, cast expression, and some kinds of literals. In other words, the usage database does not include any parameter whose expression type is none of the above, and such a parameter will not be generated as a parameter recommendation either.

While focusing Precise on these expression types, we leave out three expression types from the 7 categories in Table II, namely simple name, boolean literal and null literal. Since parameters of these expression types are recommended by the Eclipse JDT by default, Precise does not take them into account. As shown in Table II, the focal expression types of Precise take up nearly 45% of all the actual parameters. More importantly, parameters of these expression types are not recommended by existing code completion systems and are generally more difficult for developers to find or compose than simple names, boolean literals and null literals. Precise also does not deal with expression types that are not shown in Table II, because they are infrequently used and have complex structures in general.

**Rule of structural complexity.** Within the expression types specified in the first rule, there can still be some parameters that are too complex to recommend. For example, an actual parameter can be a method invocation in the form of `array[0].getFoo().getName().substring(3)`. Such complex parameters are infrequently used in practice and thus can hardly be amenable to our Precise approach which learns from existing examples. Therefore, in Precise, we restrict the two phases to the parameters (expressions) whose structural complexity is less than or equal to 3.

The *structural complexity* of an expression, `exp`, is the number of leaf nodes in the abbreviated abstract syntax tree of `exp`. The *abbreviated abstract syntax tree* of `exp`, is derived from the abstract syntax tree of `exp` by removing every node (and its sub-tree) that represents the argument list of a method invocation. Additionally, literals are treated as a leaf node, instead of being further decomposed syntactically.

This rule essentially confines Precise to the parameters whose structures are reasonably simple. According to our study on Eclipse 3.6.2, JBoss 5.0 and Tomcat 7.0, the structural complexity of nearly **95%** of the actual parameters is less than or equal to 3, which means that Precise can possibly recommend the right parameters in most cases.

### B. Building Parameter Usage Database

Following the heuristic rules, Precise first analyzes programs in a code base to build a parameter usage database. The code base is the original source from which Precise obtains its knowledge. Thus it is important to include in the code base as many programs as possible using a framework in diversified ways, in order to enable Precise to effectively recommend parameters for APIs defined in that framework.

Conceptually the parameter usage database stores the accumulated data about which parameters are used for an API in a specific context. In a recommender system for methods (e.g., [8]), it is straightforward to use the method name, (fully qualified) declaring type, and argument type list to represent a method. In contrast, Precise requires a certain degree of abstraction in representing the actual parameters in order to make the data useful in the subsequent phase.

```

1  case 1:
2  public void methodInCodeBase(){
3      Button button = new Button();
4      ...
5      this.setText(button.getAttribute());
6  }
7
8  case 2:
9  public void methodToDevelop(){
10     Button b = new Button();
11     ...
12     this.setText(?);
13 }
```

Figure 3. Example Java code, where `methodInCodeBase` is a method in the code base and `methodToDevelop` is a method in another program which is being developed.

*1) Abstract representation of parameters:* The need for abstracting the representation of parameters stems from the fact that, in different cases, some variables (or expressions) are named differently, though they essentially represent the same entity. In Figure 3, either `button` (in case 1) or `b` (in case 2) represents a reference to an instance of `Button`. Suppose the recommender learns from case 1 and tries to recommend a parameter for method `setText` in case 2, it will fail to find a match if its learning and recommendation are based on exact variable names. In this example, the most

useful information is the structure of the expected actual parameter and the type of the base variable, that is, the actual parameter should be a method invocation to `getAttribute()` and the base variable is of type `Button`.

Precise abstracts the parameter usage instances before storing them into the parameter usage database, in order to retain the essential information, while pruning unnecessary details (e.g., variable names). During the abstraction, Precise first identifies the structure of each actual parameter and categorizes the parameter into several expression types, such as literal, method invocation, and field access. Then, for each expression that is not a literal, Precise resolves the type of its sub-expressions that are local variables and replaces the variable names with their types. In Figure 3, the actual parameter used in case 1 will be represented by `Button.getAttribute()`, as the variable `button` is replaced by its type `Button`.

```

1  public class ExampleClass{
2      public void addButton(Panel panel){
3          Button b = new Button();
4          b.setVisible(true);
5          b.setAttribute(0);
6
7          panel.init();
8          panel.addElement(b);
9          //use p1 to denote b.getAttribute() here
10         panel.setAttribute(b.getAttribute());
11     }
}
```

Figure 4. More example Java code.

*2) Defining the parameter usage context:* Contextual information enables recommender systems to make recommendations according to various situations adaptively. Precise uses four static features to capture the context in which each actual parameter is used. Figure 4 shows an example program to illustrate the features. In the example, we suppose that the actual parameter of interest is the method invocation at line 11 (i.e., `b.getAttribute()`), and we use `p1` to denote it hereafter.

**Feature 1: the qualified signature of the method using the actual parameter.** This feature represents the most specific usage context for an actual parameter. In Figure 4, the raw data<sup>5</sup> of this feature of `p1` is `Panel::setAttribute(int)`.

**Feature 2: the signature of the enclosing method in which the actual parameter is used.** This feature represents the contextual information on the surrounding scope (of the method). As discussed in [8], such a feature is useful to identify the usage context in the case of overriding or implementing a method defined in a super type. Currently this feature just includes the method signature, ignoring the method's declaring type. It is a strategy to save the effort to explore type hierarchies at the cost of the accuracy of context. In Figure 4, the raw data of feature 2 of `p1` is `void addButton(Panel)`.

<sup>5</sup>The features will be transformed to facilitate the subsequent data mining.

```

{ClassA::m1(int),      outerA1, <ma1>      , <mva1, mva2>, ClassX.getInt()) }
{ClassA::m1(int),      outerA2, <ma2>      , <mva1, mva3>, ClassY.intField }
{ClassB::m2(String, int), outerB1, <mb1, mb2>, <mvb1, mvb2>, ClassZ.getStr()}
{ClassB::m2(String, int), outerB2, <      >, <mvb1, mvb2>, "str"    }

```

↓ transform

usage database							
Table: ClassA_m1_1(int)							
outerA1	outerA2	ma1	ma2	mva1	mva2	mva3	usage_index
1	0	1	0	1	1	0	21
0	1	0	1	1	0	1	14
Table: ClassB_m2_1(java.lang.String)							
outerB1	outerB2	mb1	mb2	mvb1	mvb2		usage_index
1	0	1	1	1	1		35
0	1	0	0	1	1		49

map between actual parameters and indices

```

< 21 = {ClassX, getInt(), method invocation} >
< 14 = {ClassY, intField, field access} >
< 35 = {ClassZ, getStr(), method invocation} >
< 49 = { "str", string literal} >

```

Figure 5. Example usage instances and their representations in the usage database. The usage instances are stored in two tables, namely `ClassA_m1_1(int)` and `ClassB_m2_1(java.lang.String)`.

**Feature 3: the method invocations that have happened on the variable used in the actual parameter.** For the kinds of parameters Precise focuses on, a variable  $v$  can be the base variable of a method invocation (e.g., the actual parameter is  $v.getA()$ ) or the qualifier of a field access (e.g., the actual parameter is  $v.fieldA$ ). The purpose of this feature is to capture the history of the variable  $v$  before it is used in the actual parameter. Such a history is useful to identify specific patterns of “preparing” a variable for the parameter usage. We use the method signature to denote each method invocation. Thus, in Figure 4, the raw data of feature 3 of  $p1$  is the list of methods invoked on  $b$ , that is, `<<init>(), setVisible(boolean), setAttrib(int)>`<sup>6</sup>.

**Feature 4: the method invocations that have happened on the base variable of the method invocation using the actual parameter.** Similar to feature 3, this feature is designed to capture the history of a variable, but the variable here is not used in the actual parameter, instead it is an essential part of the method invocation using the parameter. As shown in [8], such a feature can effectively represent the context of method usage. Therefore, we use this feature to include method-related context into the parameter usage context. Again we use the method signature to denote each method invocation. In Figure 4, the raw data of feature 4 of  $p1$  is the list of methods invoked on `panel`, that is, `<init(), addElement(Object), setAttrib(int)>`.

3) *Transformation and presentation of parameter usage instances:* When building the usage database, Precise first scans the code base to find every parameter usage instance. Then it extracts the feature values for each parameter usage instance, while abstracting the actual parameter. Each instance of parameter usage consists of both the feature values and the abstract parameter representation. In order to support the kNN algorithm (described in Section III-C),

<sup>6</sup>We use `<init>()` to represent the constructor of a class.

Precise transforms each parameter usage instance into a vector before storing it into the parameter usage database.

In the first place, we believe that it is generally useless to recommend an actual parameter for a method by learning from parameter usage instances of other methods or other formal parameters of the same method. Therefore, at the beginning of the transformation, we create one table in the parameter usage database for each unique formal parameter, using feature 1 (the qualified signature of the method using the actual parameter) together with the parameter’s type and position in the parameter list to generate the table name. All the actual parameters that bind to a specific formal parameter will be processed and stored in the table corresponding to that formal parameter. Then within a table, we compute a *range* for each of features 2, 3, and 4. For a specific feature, its range is the set containing all the values that have occurred in the usage instances in the same table. Each element in a range corresponds to a column in the table. When transforming a parameter usage instance, the value of a column is set to 1 if the column’s corresponding value occurs in the instance’s feature value; otherwise the column is set to 0. The last column in the table is designed for storing the actual parameter. Since the information of an actual parameter is generally too rich to be stored as plain text, we keep a map between each actual parameter and its unique index and store the indices in the last table column.

For example, Figure 5 shows the raw data of four usage instances of two different formal parameters and the structure and content of the corresponding tables in the usage database as well as the map between parameters and indices.

### C. Generating Parameter Candidates

1) *Finding Similar Usage Instances:* The kNN algorithm is a widely used machine learning algorithm whose basic idea is to classify an instance based on the “nearest” training instances (i.e., the instances whose classes/categories have already been specified). In Precise, however, the information of the nearest neighbors found by kNN is used to guide the recommendation, instead of classification. When asked for a recommendation, Precise first queries the parameter usage database using kNN to find  $k$  existing usage instances whose contexts are the most similar to the context from which the request is issued<sup>7</sup>. Then the type and structure information of these similar usage instances is used to generate a list of parameter recommendations. The details of recommendation generation will be described in Section III-C2.

For a request for parameter recommendation, the actual parameter is unknown, but its position (in the parameter list) and feature 1 is used to determine which table will be queried for its nearest parameter usage instances. To

<sup>7</sup>Similar to a parameter usage instance, a context of recommendation is also represented by a vector. However, in contrast to a parameter usage instance, a context of recommendation does not include any type or structure information of the actual parameter.



represent the context of the request, features 2 and 4 can be computed in the same way as those of the parameter usage instances in the usage database. However, since the actual parameter is yet to be recommended, it is unclear which variable will be used in the argument. Thus we have to take into account all the accessible variables for computing feature 3. For each accessible variable<sup>8</sup>,  $v$ , we will compute its history of method invocation (feature 3) and combine it with features 2 and 4 to create an individual context<sup>9</sup>, called *variable context* (and  $v$  is called the *context base* of the variable context). Besides, we will also generate an individual context called *non-variable context* only considering features 2 and 4 so as to recommend parameters whose expression types are literals. Thus, for a given request, if  $n$  variables are accessible, then there are  $n + 1$  contexts to represent the request's *possible* context. For each of the  $n + 1$  contexts, we find the  $k$  nearest instances in the usage database. In order to perform kNN queries, we first transform the request context into a vector according to the table for the formal parameter. The way of evaluating each vector element is the same as in the transformation of a parameter usage instance, except that the vector of a request context does not contain a value for the last column (i.e., the parameter index). Then we define the distance between a request context and a parameter usage instance based on their vector forms as follows:

**Definition** The distance between a request context,  $rc = \langle c'_1, c'_2, \dots, c'_n \rangle$  and a parameter usage instance,  $pi = \langle c_1, c_2, \dots, c_n, index \rangle$ , is the Euclidean distance calculated by  $distance(rc, pi) = \sqrt{\sum_{i=1}^n (c_i - c'_i)^2}$ , where  $c'_i$  and  $c_i$  ( $1 \leq i \leq n$ ) are feature values in the request context and parameter usage instance, respectively, and *index* is the parameter index.

Based on the above definition of distance, we find the  $k$  nearest neighbors for each of the  $n + 1$  context<sup>10</sup>. Then, for each context, we categorize the parameter usage instances with the same parameter index into one group. For each group, we get a summary instance that consists of the abstract parameter (corresponding to the parameter index) and the number of instances in that group (called frequency). In this way, we will get  $m$  summary instances, if there are  $m$  different parameter indices in the nearest neighbors ( $m$  may vary between different contexts).

2) *Concretizing Parameter Recommendations*: When the summary instances have been discovered from the usage database, Precise has gained some knowledge about the characteristics of the parameters used in similar contexts. As described in Section III-B1, such knowledge is abstract, in

that it mainly contains the structure and type information of the actual parameters. To provide the developer with concrete recommendations, Precise constructs parameter recommendations based on the summary instances.

During recommendation generation, Precise takes different strategies with respect to different types of expressions for different contexts. If the abstract parameter of a summary instance of a non-variable context is a literal, then Precise directly uses the literal as the parameter recommendation. Otherwise if the abstract parameter of a summary instance of a variable context contains a type name that is abstracted from a variable, Precise checks whether the context base is type-compatible with the type in the abstract parameter, and then generates a parameter recommendation by replacing the type name with the context base (i.e., an accessible variable). In this case, other segments in the parameter recommendation, such as method names or package names, are copied verbatim from their counterparts in the nearest usage instance. For example, in case 2 of Figure 3, if the abstract parameter `Button.getAttrib()` is a part of a summary instance of the variable context with context base  $b$ , Precise will generate a recommendation `b.getAttrib()`, as the type of variable  $b$  is `Button`.

After recommendation generation, we sort the recommendations (in a descending order) by the frequencies of their corresponding summary instances. Then we sort the recommendations with the same frequency in dictionary order with respect to their actual parameter expressions. Last, the sorted recommendations are presented to the user.

## IV. EVALUATION

### A. Implementation

We have implemented Precise as an Eclipse plug-in<sup>11</sup>. At the front-end, the implementation seamlessly extends the Eclipse JDT editor to present parameter recommendations. At the back-end, the implementation leverages the JDOM APIs provided by the Eclipse JDT platform to perform source code analysis, such as parsing and feature extraction, and uses Weka [13] to perform kNN queries. To use Weka, tables in the usage database are actually stored as ARFF<sup>12</sup> files, and the map between index and actual parameter information is stored in XML files. These two parts, which are correlated by the indices of actual parameters, play the role of the usage database in Precise.

### B. Objective Experiment

**Performance measure.** Using the implementation, we have performed an objective experiment to evaluate the usefulness of Precise. We impose a constraint on the number of parameter candidates, that is, we only check the top 10 candidates if there are more than 10 of them. The reason

<sup>8</sup>Accessible variables include local variables, formal parameters of the enclosing method, and fields.

<sup>9</sup>Features 2 and 4 are common for all the accessible variables.

<sup>10</sup>We first set  $k$  to 1 and increment  $k$  until the number of concrete recommendations reaches a threshold or all the instances in the usage database are used. Currently the threshold is set to 10.

<sup>11</sup>Available via <http://stap.sjtu.edu.cn/~chengzhang/precise/>

<sup>12</sup>The native input format of Weka.

is that the user will have a cognitive context switch [19] if she has to scroll down the list to check the recommendations lower than 10. It is reasonable to assume that all the top 10 recommendations will attract equal attention from the user, because they are presented in a short list that can be quickly checked at the first glance. In addition, by controlling the total number of recommended parameters, we can evaluate Precise’s usefulness by checking how often it is able to provide useful recommendations, without being concerned about how many recommendations it generates each time<sup>13</sup>.

It is straightforward to determine whether Precise is successful for a specific request by checking whether the list of parameter recommendations includes the expected actual parameter. Thus we use  $Recommendations_{made}$  to denote the total number of times that Precise tries to recommend a list of parameters and  $Recommendations_{successful}$  to denote the number of times that the list includes the expected parameter. Then we use the recall (defined as below) to represent Precise’s performance of parameter recommendation.

$$recall_{successful} = \frac{Recommendations_{made} \cap successful}{Recommendations_{made}}$$

However, as in the case shown in Figure 1, even if the recommendations fail to include the exact actual parameter, they can still provide useful information, such as which class should be used as a part of the parameter. More specifically, a recommended parameter is said to be *partially successful*, if it correctly indicates 1) the type of the base variable of a method invocation, 2) the qualifier of a qualified name, or 3) the method name of a method invocation. In order to take into account such cases, we use  $Recommendations_{useful}$  to denote the number of times that the recommendation list provides either completely or partially successful information with respect to the actual expected parameter. Then we define another recall to measure Precise’s performance from this viewpoint. In our evaluation, we study both  $recall_{successful}$  and  $recall_{useful}$ .

$$recall_{useful} = \frac{Recommendations_{made} \cap useful}{Recommendations_{made}}$$

**Subjects and validation strategy.** Since the main purpose of Precise is to recommend parameters to facilitate the use of framework APIs, it is reasonable to evaluate Precise on individual frameworks separately. In our experiment, we focus on the SWT framework [6] and use all the projects (using SWT) in the Eclipse code base as the subjects. We choose the SWT framework because it is widely used in open source projects, providing Precise with a large code base (e.g.,

<sup>13</sup>It is generally useless to recommend a large number of parameters, even if the right parameter is included, because the user is likely to prefer composing the parameter by herself, rather than find the needle in a haystack.

Eclipse). Moreover, SWT is a specialized framework for GUI development. Compared with common libraries, such as JDK collections, SWT is less familiar to developers and has more specific patterns of parameter usage. Therefore, developers are more likely to benefit from an enhanced code completion system for the SWT framework.

We take the strategy of 10-fold cross validation<sup>14</sup> to split the parameter usage instances (in the Eclipse code base) into two parts from which the larger part is used for building the usage database and the smaller one is used as the test set. We split the parameter usage instances based on class, that is, all the parameter usage instances occurring in a Java class were used together in either the test set or the training set. During the experiment, each actual parameter (in the test set) of an API defined in SWT will be used to test Precise. When being used as the test instance, the actual parameter is “hidden” and a request is issued to Precise for recommendations. This process simulates the scenario that a developer has already selected a method to call and is about to find an actual parameter for the method. The recommendations for the request are checked against the actual parameter to see whether they are successful or useful.

Table III  
RESULTS OF AN OBJECTIVE EXPERIMENT ON PRECISE.

No.	#Param	#Req	#Hits	#useful	$r_{successful}$	$r_{useful}$
1	3095	978	546	89	56%	65%
2	3785	1154	599	157	52%	66%
3	3252	984	516	123	52%	65%
4	2967	931	495	102	53%	64%
5	2932	950	443	107	47%	58%
6	3504	1094	575	105	53%	62%
7	3030	928	562	110	61%	72%
8	3191	994	513	88	52%	60%
9	3275	1043	530	106	51%	61%
10	3313	996	501	140	50%	64%
Avg.	32344	10052	5280	1127	<b>53%</b>	<b>64%</b>

<sup>14</sup>As we use 10-fold cross validation, there are 10 rows of data in total. Among the columns, ‘#Param’ means the number of actual parameters used in the test set; ‘#Req’ means the number of recommendation requests issued to Precise; ‘#Hits’ means the number of times when the list of recommendations contain the expected actual parameter;  $r_{successful}$  and  $r_{useful}$  stand for  $recall_{successful}$  and  $recall_{useful}$ , respectively.

**Results.** Table III shows the experimental results. We can see that the average  $recall_{successful}$  and  $recall_{useful}$  of Precise are 53% and 64%, respectively. This means that Precise is able to provide useful recommendations most of the time and sometimes the right parameter is included in the list of recommendations. It is important to note that the kinds of parameters recommended by Precise are not recommended by the default Eclipse JDT and they are generally more complex and difficult to find or compose. Therefore, these can be viewed as promising results.

<sup>14</sup>More specifically, the whole data set is evenly divided into 10 subsets at random. In each iteration of validation, one subset is used as test data, while the other nine subsets are used as training data. Ten iterations are performed so that each subset is used as test data once.

Although  $recall_{successful}$  and  $recall_{useful}$  are direct metrics to measure Precise’s performance in parameter recommendation, they may not be suitable to reflect the user experience of Precise, since the list of recommendations presented to the user also contains the recommendations generated by the default Eclipse JDT. From the perspective of users, we are more interested in how much improvement Precise has made to the default Eclipse JDT code completion system. The default Eclipse JDT code completion system generates its parameter recommendations based on simple type-based rules. As discussed in Section III-A, the recommendations include limited (simple) types of expressions. By default, the JDT code completion system is activated every time a parameterized method is selected by the developer. Whenever the expression type of the expected parameter is in the scope of the JDT code completion system, it will certainly be included in the list of recommendations (although there may be more than 10 parameters recommended). However, due to its limitation on expression type, the JDT code completion system will certainly fail if the expected parameter is complex. As shown in Figure 6, the  $recall_{successful}$  is 47%. As the combination of the default JDT and Precise, the enhanced code completion system can achieve a  $recall_{successful}$  of 65%, which is about 38% higher than the original JDT system. Moreover, the enhanced system can achieve a  $recall_{useful}$  of 72%, indicating that it can often provide useful recommendations.

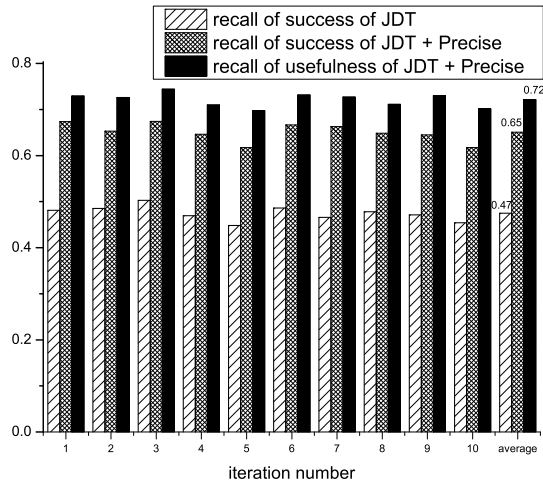


Figure 6. Results of the objective experiment on Eclipse JDT and Precise.

Since the experiment is focused on the top 10 recommendations, the effectiveness of Precise’s ranking strategy can affect the results significantly. As shown in Figure 7, 93% of the successful recommendations are placed in the top 10 of the list. In comparison, only 49% of the partially correct recommendations are ranked top 10. By analyzing the result data, we found that a large number of low-ranked partially correct recommendations are generated for some common methods (the limitation of Precise on common methods will

be discussed shortly). Another indication of Figure 7 is that there may be large room for improvement in  $recall_{useful}$  by designing better ranking strategies.

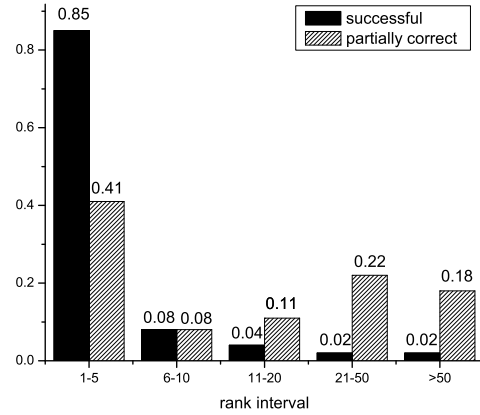


Figure 7. Rank distributions of recommendations.

A concern on the usability of Precise is its runtime, because it involves user interactions. We conducted a special experiment to study this issue. In the experiment, we measured the runtime of the first phase (usage database building) on a powerful server with a 2.33GHz quad-core CPU and 16GB memory, while we measured the runtime of the second phase (recommendation generation) on a desktop with a 1.60 GHz dual-core CPU and 2GB memory. We took this strategy because the first phase is supposed to be quite time-consuming and will generally be performed only once on a large code base, while the second phase is mostly performed on the user’s workstation which is usually less powerful than a server. As expected, most of the runtime (792.7 seconds on average) was taken by usage database building. In comparison, the second phase took little runtime (0.231 second on average), which is presumably negligible. Using a pre-built usage database, the user can only perceive the delay of the second phase. Therefore, we believe that Precise can be seamlessly integrated into the default Eclipse JDT code completion system. In our user study, we have designed a question to further investigate this problem.

**Limitations.** An important assumption for Precise’s effectiveness is that it can learn usage patterns from the code base to guide its recommendation process. However, there exist a number of APIs that are widely used in various contexts. These APIs (e.g., `get(Object)` in `java.util.Map`) are so common that their actual parameters are generally too diversified for Precise to learn any usage pattern. Although we focused our experiments on a specialized framework, SWT, there are still a number of common methods defined. A typical example is the method `setText(String)` defined in classes `Button`, `Label`, etc. The method is designed to set the text that will be displayed on a widget. It is imaginable that numerous strings will be used as the



actual parameters of `setText` in a functional GUI application. As a result, Precise had extremely bad performance for this method, usually recommending a large number of string literals and variables without being successful. In the experiment, `setText` takes up 32% of the requests for recommendations and the *recall<sub>successful</sub>* and *recall<sub>useful</sub>* of Precise for `setText` are 21% and 24%, respectively. However, we argue that once developers have some basic knowledge about such methods, they generally will not expect a code completion system to provide parameter recommendations, because they probably know that they should take full control of the parameter usage in this case.

As a learning-based approach, if Precise has never learnt a certain usage pattern from the existing code base, it has little chance of recommending the right (or useful) parameters. In fact, this inherent limitation is a major reason for Precise’s failures. During the experiment, the expected actual parameter is sometimes a method or field that is used very locally (e.g., a private field can only be used in its declaring class). Meanwhile, we split the parameter usage instances based on class in the 10-fold cross validation. When the usage of such a method or field are concentrated in a small number of classes in the test set, they are out of the reach of Precise. Another similar reason for failure is that some methods have never been called in the training set. In this case, when requested to recommend parameters for such a method, Precise is unable to find a corresponding table in the usage database and thus provides no recommendations. In order to overcome the difficulties in recommending unseen parameter usage, we are planning to enhance Precise with some generative techniques (e.g., [12]) in our future work.

### C. User Study

Besides the objective experiment, we have conducted a user study to further investigate the usefulness of Precise. In the user study, we invited 8 participants to use the Precise implementation to finish two small programming tasks. The participants are students with more than 3 years experience of Java programming and at least 2 years experience of Eclipse on average. Thus they are familiar with the default Eclipse JDT code completion system. The two programming tasks are adapted from the examples of using SWT framework<sup>16</sup>. We deleted several statements from the original examples, leaving the skeletons of the programs. In this way, the participants can concentrate on finishing the programs using APIs from SWT, without making design-level efforts, such as creating new interfaces or classes. We set a loose time limit of 90 minutes for the participants, that is, they were encouraged to give up after trying for 90 minutes if they found it too difficult to finish the tasks correctly. However, they could choose to go on with the tasks. The

purpose of the time limit is to avoid wasting time when the participants have already experienced Precise sufficiently.

After doing the programming tasks, each participant was asked to fill in a questionnaire that consists of five questions. For each question, a participant first chose a score from 5 (strong agreement), 4 (weak agreement), 3 (median), 2 (weak disagreement), and 1 (strong disagreement), and then gave a detailed explanation of the choice. The questions and the summary of the answers are described as below.

**Question 1: Did Precise propose the right parameters?** (average score: 3.875, median score: 4) This question is designed to confirm that the users can mostly recognize the successful recommendations. The resulting scores and the explanations show that the participants were able to identify the right parameter in most cases. This was partly due to the short list of recommendations provided by Precise. The scores also indicate that the participants perceived the effectiveness of Precise in accordance with the *recall<sub>successful</sub>* (65%), that is, it is useful, although not perfect.

**Question 2: When you could not find the right parameter in the recommendations, did you feel Precise still gave some hints?** (average score: 3.875, median score: 4) This question is designed to investigate whether the partially correct information can be indeed useful to users. We almost get the same positive answers as in Question 1. A participant said: “*Some of the parameters are “dot-outcome”, and I really feel it’s convenience*”. While confirming the partial usefulness, the second answer also suggests that the heuristic rules of Precise probably choose the right expression types (e.g., method invocations and qualified names) to focus on.

**Question 3: Did Precise correctly rank the parameters?** (average score: 3.5, median score: 3) This question is designed to check our ranking strategy and our constraint on the number of recommendations. As shown by the scores, the participants did not think Precise ranked the recommendations well. We investigated the explanations and found that some participants just did not pay attention to the ranking. One of them simply stated: “*I didn’t notice this feature (parameter ranking)*”. It suggests that some developers may focus on finding the right parameter in the list, concerning little about the ranking (of the recommendations that are visible in the list). Another participant said: “*I am not quite clear about the rank; I hope it provided more easy-understood explanations*”. The ranking strategy is mainly based on similarity of context and frequency of usage and its purpose is to put the most promising recommendations at the top of the list. However, users probably wonder why such a sophisticated ranking (instead of dictionary order) is used, especially when they fail to find the right parameters and try to use the partial information. We are planning to improve this aspect of Precise in our future work.

**Question 4: Did Precise speed up your development compared to the default Eclipse code completion?** (average score: 3.375, median score: 4) By asking this question,

<sup>16</sup>One task is to implement a simple panel with various fields and texts, while the other is to implement a simple browser. The adapted programs are available via <http://stap.sjtu.edu.cn/~chengzhang/precise/>

we attempted to investigate whether the improvement by Precise can be noticeable in the whole Eclipse system. One participant said: *“It is helpful, especially when Eclipse fails to give the right parameter”*. The answer confirms that it is necessary to extend the default Eclipse JDT code completion system. Although the answers are generally positive, they are far from conclusive, because various factors may affect the final results. One participant stated: *“I don’t know why it nearly didn’t propose, and this cannot be judged”*. After careful checking of the participant’s program, we found that she mostly failed to find the right methods to call. As a result, Precise was not activated as expected. We believe that the integration of novel code completion techniques (e.g., API ranking [8] and Precise) could improve the existing system more significantly than individual techniques.

**Question 5: Is Precise well integrated into Eclipse?** (average score: 4.28, median score: 4) The main purpose of this question is to investigate whether the runtime overhead of Precise is acceptable in practical settings. Among the seven participants who answered this question, three of them chose the score of 5. Four participants expressed the feeling that Precise worked as a natural part of the Eclipse JDT. One participant said: *“Maybe good, when it proposed, the eclipse did not slow down.”* The answers have confirmed that Precise does not cause perceivable slowdown when proposing recommendations.

In summary, the user study has confirmed the usefulness of Precise shown by the objective experiment. However, it has also revealed an inadequacy of Precise: its ranking strategy lacks explanations and better integration with other techniques is needed.

## V. RELATED WORK

**Recommender System.** Recommender systems have a wide range of applications in software engineering to improve productivity and reliability. Giger et al. [11] built a recommender system to predict the fix time of a reported bug, helping developers to select the proper issues to be fixed. Kpodjedo et al. [16] proposed a recommender system based on error correcting graph matching and random walks to assess the criticality of classes in the system. In Precise, we attempt to improve programmers’ productivity by recommending parameters through a code completion system.

**API recommendation.** Bruch et al. [8] presented Intelligent Code Completion Systems (ICCS) which learns from existing programs to generate API usage recommendations. The ICCS searches the code repositories for API calls under the similar context and generates proposals using a customized kNN algorithm. Similarly Precise uses data mining techniques to make recommendations. However, while ICCS makes recommendations exclusively on method calls, Precise aims to predict actual parameters for method calls. Therefore, Precise can be seen as an extension to ICCS. Robbes and Lanza [19] propose to use program history to

improve code completion systems. They model the program history as a sequence of changes and invent a series of code completion algorithms mainly based on the method-level and class-level changes. The aim of their work is the same as that of [8], thus different from ours. However, the idea of using program history may also be promising to improve parameter recommendation. Another approach [14] enhances the code completion system by introducing new features such as grouping, sorting and filtering. Although these features enable the recommender system to further prune the number of candidates, the sorting algorithm is based merely on frequency. Consequently the proposals made for the same object are invariant regardless of the context. Precise handles this problem by adopting machine learning algorithms to the recommender system. Other techniques [17], [10], [15], [18] have also been proposed to facilitate the use of API. Different from Precise, they are not focused on parameter recommendation and code completion.

**Code search.** Code search engines enable developers to learn the usage of unfamiliar APIs by providing code examples. Bajracharya et al. [7] create SAS based on the Sourcerer infrastructure that searches large code bases for API usage examples. SAS applies a grouping and indexing algorithm to search the repository and shows relevant code snippets when a developer queries about a specific API usage situation. Although these features of SAS provide better facilities for API understanding than document browsing, SAS still requires developers to understand the APIs and pick from the list of search results the right methods or arguments to be placed in their particular context. In contrast, with Precise, this task is done by the kNN algorithm automatically. Recommendations are generated and proposed to the developer through the code completion system seamlessly, freeing them from learning the rationales of the relevant APIs. Besides the text search used by SAS, structural information of program source code is also used in code search engines, such as SNIFF [9]. SNIFF is more relevant to Precise in that it makes use of type information to refine its result set. Other approaches (e.g., [22]) also leverage search engines to facilitate the reuse of frameworks. Compared with Precise, these search-based techniques generally require users to provide search keywords and the search probably incurs perceivable waiting time.

## VI. CONCLUSIONS AND FUTURE WORK

We have proposed Precise to automatically recommend actual parameters for APIs. By learning usage patterns from existing programs, Precise provides useful parameter recommendations with a satisfactory recall, showing its ability to improve the state-of-the-art code completion systems. In our future work, we are planning to study how to use program evolution information to improve Precise. In addition, we will try to use generative techniques to recommend parameter usage that are unseen in the code base.

## REFERENCES

- [1] Apache tomcat. <http://tomcat.apache.org/>.
- [2] Eclipse documentation. <http://help.eclipse.org/helios/index.jsp>.
- [3] Eclipse project. <http://www.eclipse.org/>.
- [4] The Java language specification, third edition. [http://java.sun.com/docs/books/jls/third\\_edition/html/j3TOC.html](http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html).
- [5] Jboss application server. <http://www.jboss.org/jbossas/>.
- [6] SWT: The standard widget toolkit. <http://www.eclipse.org/swt/>.
- [7] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, SUITE '09*, pages 1–4, Washington, DC, USA, 2009. IEEE Computer Society.
- [8] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering, ESEC/FSE '09*, pages 213–222, New York, NY, USA, 2009. ACM.
- [9] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for Java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, FASE '09*, pages 385–400, Berlin, Heidelberg, 2009. Springer-Verlag.
- [10] Barthélemy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 481–490, New York, NY, USA, 2008. ACM.
- [11] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE '10*, pages 52–56, New York, NY, USA, 2010. ACM.
- [12] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and Practice of Declarative Programming, PPDP '10*, pages 13–24, New York, NY, USA, 2010. ACM.
- [13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.
- [14] Daqing Hou and David M. Pletcher. Towards a better code completion system by API grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE '10*, pages 26–30, New York, NY, USA, 2010. ACM.
- [15] David Kawrykow and Martin P. Robillard. Improving API usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 111–122, Washington, DC, USA, 2009. IEEE Computer Society.
- [16] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, and Giuliano Antoniol. Not all classes are created equal: toward a recommendation system for focusing testing. In *Proceedings of the 2008 international workshop on Recommendation Systems for Software Engineering, RSSE '08*, pages 6–10, New York, NY, USA, 2008. ACM.
- [17] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI '05*, pages 48–61, New York, NY, USA, 2005. ACM.
- [18] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10*, pages 302–321, New York, NY, USA, 2010. ACM.
- [19] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Eng.*, 17:181–212, June 2010.
- [20] Martin P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Softw.*, 26:27–34, November 2009.
- [21] Jeffrey Stylos, Brad A. Myers, and Zizhuang Yang. Jadeite: improving API documentation using usage information. In *Proceedings of the 27th international conference Extended Abstracts on human factors in computing systems, CHI EA '09*, pages 4429–4434, New York, NY, USA, 2009. ACM.
- [22] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering, ASE '07*, pages 204–213, New York, NY, USA, 2007. ACM.
- [23] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.